加餐｜李程远：谈谈 eBPF 在云原生中的纵与横

2022-03-07 李程远

《eBPF 核心技术与实战》 课程介绍 >

讲述：山荣
时长 13:59 大小 12.82M

你好，我是李程远。很高兴受邀来到这门课做一期分享。如果你之前学过极客时间上的另一个专栏《容器实战高手课》，应该会对我比较熟悉。

今天想跟你聊的，是一些我自己关于用 eBPF 进行系统黑盒诊断的思考，特别是在云原生平台上的应用。从 2014 年进入到 Linux 内核以来，eBPF 一直是 Linux 内核中最火的领域。作为 eBPF 的三大应用领域之一，在 Linux 内核的追踪 / 调试中，特别是在云平台来定位一些复杂问题时，eBPF 已经处于不可替代的地位了。

在《容器实战高手课》的一篇加餐里，我也简单介绍过 eBPF 这个技术。当时我给了学们一个例子，通过它看了如何用 eBPF 来定位我们生产环境中的数据包网络延时偶尔增大的原因。最近，我又碰到一个生产环境中的网络问题，仍然还是依靠 eBPF 程序的帮助，定位到了原因。今天，我就先跟你分享下这个问题的具体情况，以及用 eBPF 定位原

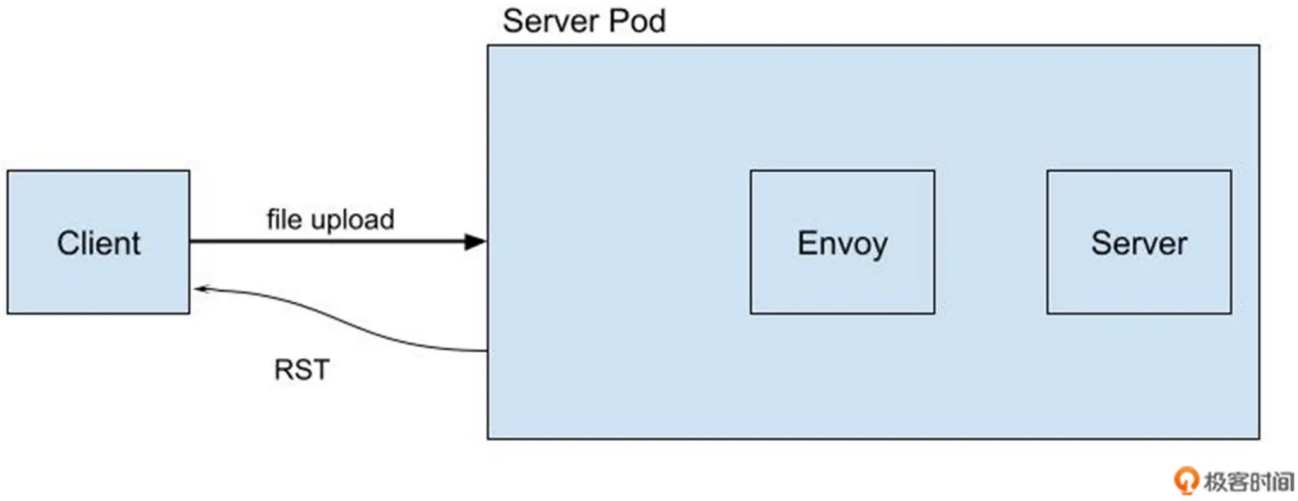
因的过程。然后，我会从这个例子出发，聊聊 eBPF 程序可以怎样更好地在云原生平台上应用。

一个例子：用 eBPF 解决生产环境中的网络问题

关于遇到的这个问题，先来说一下我看到的现象。

把线上的问题简化之后，我看到 Client 向一个 Server Pod 里的服务上传数据的时候，偶尔连接会发生中断。通过对 Server Pod 所在的宿主机节点上 tcpdump 数据包的抓取，我们会看到，从 Server Pod 向 Client 发送了一个 TCP RST(reset) 数据包之后，上传数据的连接就中断了。

显然，我们下一步要做的就是**找到这个 TCP RST 发送的原因**。不过 TCP RST 发送的原因有很多种，同时，在我们的 Server Pod 里已经注入了 Envoy sidecar 容器，用了 service mesh 之后，这样 Pod network namespace 里的网络拓扑的复杂度也提高了。这个问题大致的示意图如下：



在这种情况下，我们想快速找到这个 TCP RST 发送的原因，就需要借助 eBPF 的诊断程序。一个最直接的思路就是找到内核中的发送 TCP RST 数据包的函数，也就是 tcp_v4_send_reset() 和 tcp_send_active_reset() 这两个函数，然后用 eBPF 的 kprobe hook 来追踪这两个函数的调用。

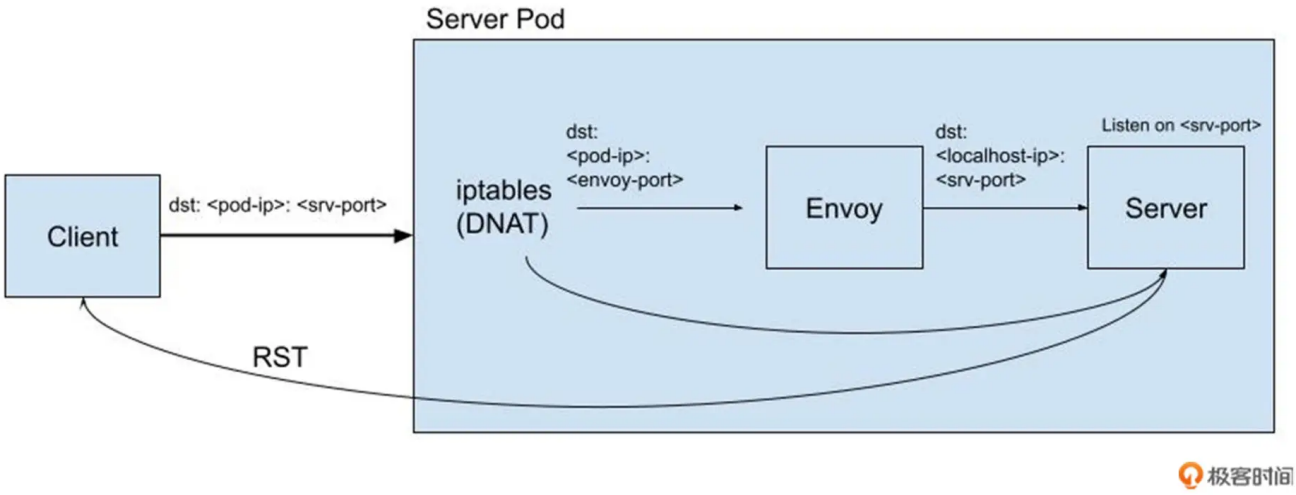
通过这些函数的输入参数，我们可以知道 TCP RST 发送前收到的数据包的内容、对应 socket 的状态。我们还可以知道这些被追踪函数的调用关系。

在运行 eBPF 的诊断程序之后，我们发现了两点：

第一，发出 TCP RST 数据包所对应的 socket 是处于 TCP_LISTEN 状态的，并且 socket 的监听端口是服务端口 srv-port，那么这个显然是 Server 程序的 LISTEN socket。

第二，内核在发出 TCP RST 数据包之前收到的数据包，其目标地址是:。

整个场景的数据包流程图如下：



这里，我们结合数据在 Server Pod 里的流向来看下：当 Envoy sidecar 容器注入到 Server Pod 后，会通过 iptables(DNAT) 方式将数据流先重定向到 Envoy, 然后再由 Envoy 将数据发送给 Server。

结合上面 eBPF 诊断程序的两点发现，我们就会一下子意识到问题所在：某一个数据包没有经过 iptables 的 DNAT 操作，绕过了 Envoy 直接发送到了 Sever，从而触发了一个 TCP RST 被 Client 接收到。

分析到这里，其实我们基本上已经知道产生 TCP RST 的原因了。那么，iptables 为什么没有对某个数据包做 DNAT 呢？这是因为，在 Linux 内核的 conntrack 机制里，如果收到了乱序的包，在缺省配置的情况下（这里提示下，可以去了解一下内核 ip_conntrack_tcp_be_liberal 这个参数），就是会放过这个包而不去做 NAT 的，这是一个很常见的问题了。

如何在云原生平台上更好地使用 eBPF 程序？

从上面这个问题的调试诊断过程，我们可以看到 eBPF 在云原生环境中进行黑盒调试的威力。不过，如何让 eBPF 程序在云原生平台上更好地被使用呢？按我目前的理解，要从两个维度来考虑，一个是纵向的，另一个是横向的。

纵向的深入

那什么是纵向呢？简单来说，就是需要不断地加深对内核的理解。

为什么这么说呢？如果我们来看 eBPF 程序的编写方式，其实它看起来是非常非常固定的。eBPF 程序的核心概念，也就是 eBPF program、eBPF map 和 helper function 这些。如果去看 BCC 下面的工具代码，你也会发现，每个工具的程序代码结构几乎都是差不多的。这一切都说明，eBPF 其实是很“简单”的。

但是，真的是很简单吗？如王阳明所说，“不行不足谓之知”，如果你动手实践一下，让自己从零开始写一个 eBPF 的程序，可能一上来就会发现事实并非如此。光是弄清楚如何写好它的 Makefile，就需要花费一番功夫。至于后面的代码编写，你会发现代码量不大，很多代码还可以依葫芦画瓢，参考网上已有的代码，运气好的话可能会编译执行通过了。但是，一旦编译或者代码运行中出现问题，需要解决的问题其实就只有几行代码，可花费的功夫会远远超过编写整个代码的过程。

为什么会这样呢？因为在编写 eBPF 代码的过程中，你需要运用到 Linux 内核的基本实现原理。我在这里举几个例子吧。

比如，如果我们需要追踪一个内核函数，就会发现有好几种 eBPF program 的类型可以选择，每一种 eBPF program 的输入参数还都是不一样的。如果我们需要追踪内核函数 __set_task_comm（），想看看哪个进程的进程名字被修改了，可以用 kprobe 的方式来追踪它，也可以用 tracepoint 的方式，还能通过 raw_tracepoint 的方式来追踪。

在使用 kprobe eBPF program 的时候，它的输入参数是 struct pt_regs；在使用 tracepoint eBPF prgoram 的时候，它的输入参数是一个自定义的结构 struct task_rename；而到使用 raw tracepoint 方式的时候，输入参数又变成了 struct bpf_raw_tracepoint_args。不知道你在使用 eBPF program 的时候，有没有想过这个问题：为什么这些参数是不一样的？

复制代码

```
void __set_task_comm(struct task_struct *tsk, const char *buf, bool exec)
{
        task_lock(tsk);
        trace_task_rename(tsk, buf);
        strlcpy(tsk->comm, buf, sizeof(tsk->comm));
        task_unlock(tsk);
        perf_event_comm(tsk, exec);
}
```

复制代码

```
SEC("kprobe/__set_task_comm")
int prog(struct pt_regs *ctx)
{
  return 0;
}

/* from /sys/kernel/debug/tracing/events/task/task_rename/format */
struct task_rename {
__u64 pad;
__u32 pid;
char oldcomm[16];
char newcomm[16];
__u16 oom_score_adj;
};
SEC("tracepoint/task/task_rename")
int prog(struct task_rename *ctx)
{
return 0;
}


SEC("raw_tracepoint/task_rename")
int prog(struct bpf_raw_tracepoint_args *ctx)
{
return 0;
}
```

这个问题的答案，其实还是要从内核中 kprobe、tracepoint 等内核追踪技术里得到。比如 kprobe ，是通过把当前地址上的指令替换成 int3 中断指令来实现的。而 Linux 内核会在中断发生的时候，把当前各个寄存器的内容写入到栈内存中，而结构 pt_regs 就是用来描述这块栈内存里的内容的。因此，kprobe eBPF program 的输出参数就是 struct pt_regs，根据 Linux ABI，我们就可以知道，被追踪函数的第一个参数的值可以从 pt_regs->di 里得到了。

再来看 tracepoint。对于每一个 tracepoint，它都是内核中的一个特别的函数。比如上面例子中的函数 trace_task_rename()， 函数的实现是通过 tracepoint 的一个固定模板（宏）定义的。通过 bpf tracepoint 的 commit，我们看到，在 tracepoint 被调用的时候，TP_STRUCT__entry 里的变量会被赋值，tracepoint eBPF program 的输入参数就是每个 tracepoint 中 TP_STRUCT__entry 的内容。tracepoint 宏定义如下所示：

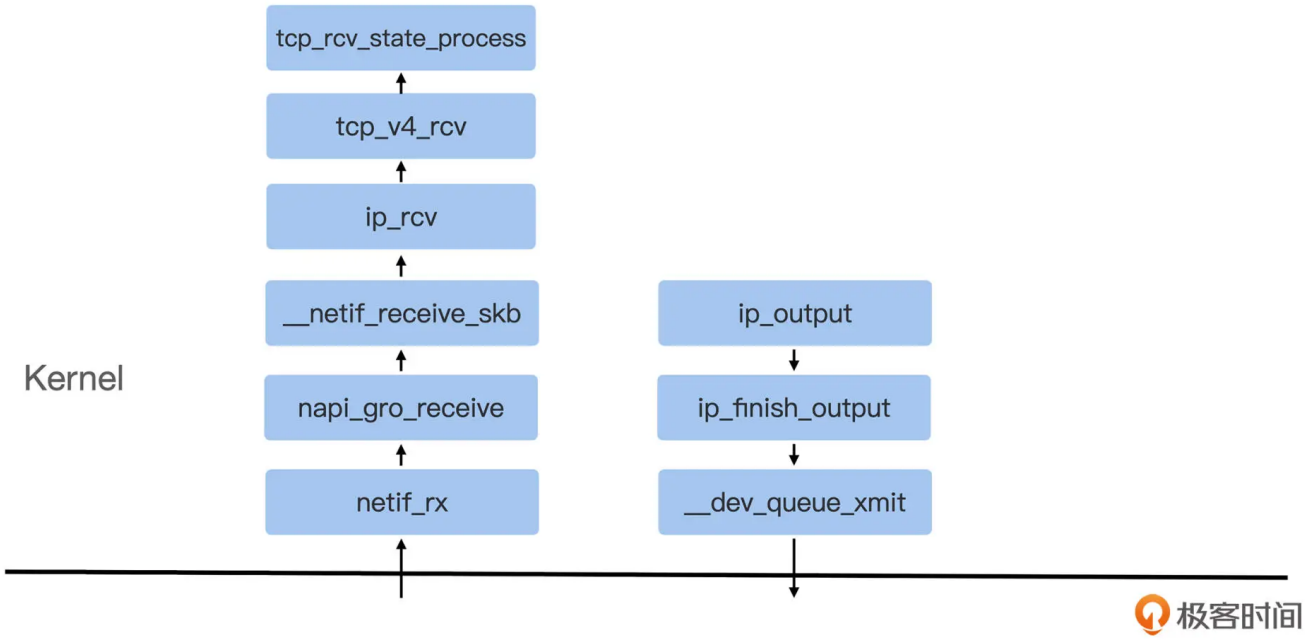
复制代码

```
TRACE_EVENT(task_rename,

        TP_PROTO(struct task_struct *task, const char *comm),

        TP_ARGS(task, comm),

        TP_STRUCT__entry(
                __field(        pid_t,  pid)
                __array(        char, oldcomm,  TASK_COMM_LEN)
                __array(        char, newcomm,  TASK_COMM_LEN)
                __field(        short,  oom_score_adj)
        ),

        TP_fast_assign(
                __entry->pid = task->pid;
                memcpy(entry->oldcomm, task->comm, TASK_COMM_LEN);
                strlcpy(entry->newcomm, comm, TASK_COMM_LEN);
                __entry->oom_score_adj = task->signal->oom_score_adj;
        ),

        TP_printk("pid=%d oldcomm=%s newcomm=%s oom_score_adj=%hd",
                __entry->pid, __entry->oldcomm,
                __entry->newcomm, __entry->oom_score_adj)
);
```

最后看 raw tracepoint。这是 eBPF 引入的 raw tracepoint program ，其目的就是访问 tracepoint 定义里的 TP_PROTO 参数。比如在我们当前的这个例子里，用了 raw tracepoint 就可以访问到 task_struct *task 参数了，可以得到 task_struct 里的所有信息，而不是 TP_STRUCT__entry 里的那几个变量了。

上面的几个例子说明，编写 eBPF 程序首先需要理解 Linux 内核的追踪机制的实现。而在使用 eBPF 定位具体内核问题的时候，就更需要理解内核对应部分的代码了。比如，如果我们要诊断网络问题，很自然地，我们需要知道一个网络数据包从网卡驱动进入到内核网

络协议栈的主要函数，然后对这些函数进行追踪。下面是内核网络协议栈数据包的接收和发送主要函数图：



再举一个例子：在云原生平台上，对内核函数的追踪往往需要我们知道这个函数的调用是在哪个 namespace 里，这样就可以知道是哪个容器触发的操作。大部分的 namespace 都可以通过进程控制块 task_struct{} 里的 nsproxy 结构定义得到，不过 pid namespace 有点特殊了： nsproxy 里的 pid_ns_for_children 并不是代表了进程当前所属的 pid namespace，进程真正的 pid namespace 的值需要从进程控制块的 pid->numbers[pid->level].ns 中得到。

复制代码

```
struct nsproxy {
  atomic_t count;
  struct uts_namespace *uts_ns;
  struct ipc_namespace *ipc_ns;
  struct mnt_namespace *mnt_ns;
  struct pid_namespace *pid_ns_for_children;
  struct net     *net_ns;
  struct time_namespace *time_ns;
  struct time_namespace *time_ns_for_children;
  struct cgroup_namespace *cgroup_ns;
};
```

从上面的这些例子中，我们可以看到：如果以一棵树来比喻的话，各种 eBPF 的工具程序只是树的枝叶，而树的根本还是 Linux 内核各种实现机制。因此，在学习 eBPF 的时候，

还是要在根本上多花功夫，这也是我这里说的“纵向”功夫。

横向的应用

说完了纵向的问题，那么你肯定想问是不是还有横向的问题。那么什么是横向的问题呢？

纵向的问题是在一个节点上的深入，而横向的问题是在云原生平台上，如何对几千几万台机器使用 eBPF。在生产环境的云原生平台上使用 eBPF，一般有两种场景：

第一种是针对生产环境中的特定问题做诊断。我们在这一讲开头介绍的 Client 收到异常 TCP RST 的那个问题，就是属于这种场景。

第二种是通过 eBPF 来采集内核相关的参数，作为云平台常规监控数据的一部分，进行长期的收集。

对于第一种场景，在几千几万台机器中使用 eBPF 来定位问题，和只在一台机器上运行 eBPF 程序还是有很大不同的，它需要考虑到下面两个问题。

第一个问题是**权限的问题**。在生产环境的云平台上，有 root 权限登录宿主机的同学很少，用户的 pod/container 绝大多数都是 non-privileged，而运行 eBPF 程序又需要有 privileged 的权限。这样就产生了一个很大的矛盾： 如果用户的 pod 出现网络问题，希望使用 eBPF 程序来做深入的诊断，而用户自己根本就没有权限，这样一旦发生问题，只能依靠有 root 权限的同学登录到宿主机上去执行 eBPF 程序。而有 root 权限的同学寥寥无几，这也会成为瓶颈，这样就把 eBPF 在云平台上的使用门槛不必要地拉高了。

第二个问题是**多节点操作的问题**。在生产环境的云平台上，要诊断一个问题，往往需要同时在多个节点上执行程序或者收集数据。比如要解决网络延时的问题，就需要在 client pod 宿主机、software LB 宿主机、server pod 宿主机上同时对一个流的数据包进行追踪。类似的多节点诊断操作在生产环境的云平台上常常会发生，如果每次都是手动到各个节点上去执行 eBPF 程序，然后汇总数据，那么工作效率也是很低下的。

要解决上面的两个问题，我们需要在云平台上建立一个运行 eBPF 诊断程序的框架。这个框架至少要包含三个部分：

第一部分是在每个宿主机上的 agent。这个 agent 可以用来运行各种 eBPF 诊断程序；

第二部分是一个 controller。这个 controller 用于接收到用户诊断的命令，然后协调相关宿主机里的 agent 来运行 eBPF 程序并且汇总结果；

第三部分是一个用户界面，包含用户认证、权限控制，让用户输入诊断指令并且输出结果。Brendan Gregg 在他的一篇文章 eBPF observability blog 里也提到了在 Netflix 使用的 eBPF 诊断框架。

在云平台上有了 eBPF 诊断的框架之后，它不仅可以帮助解决我们前面说的第一种场景中碰到的问题，其实也可以为第二种场景，也就是监控方面提供服务。在每个宿主机上运行的 agent 不但可以接收指令来执行特定的 eBPF 程序，也可以不断输出 metrics 与现有的监控平台相结合。比如，可以使用 eBPF_exporter 把一些内核相关的 metrics 不断地输出。开源项目 pixie 也提供了在云原生平台上的一个比较完整的 observability 框架和工具，底层的 agent 也是基于 eBPF。

总结

今天的分享就到这里了，我们最后来小结一下。

在云原生平台上，eBPF 的诊断程序可以帮助我们发现很多深层次 Linux 内核相关的问题，是云原生环境中黑盒调试的利器。要写好 eBPF 的诊断程序，我们需要花更多的功夫在 Linux 内核各种实现机制的研究上。而要把 eBPF 诊断程序更好地运用到云平台上，我们需要设计和实现一个框架。

感谢你看到这里，如果今天的内容让你有所收获，欢迎把它分享给你的朋友。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 未来可期 | 邀你与 eBPF 共赴一场技术革新之约

精选留言 (1)

写留言

莫名 神

2022-03-07

一纵一横，直击要点。 李老师是 eBPF 实战派高手，深谙底层技术原理，《容器实战高手课》加餐篇可见一斑。曾受益匪浅，再次表达感谢与钦佩。

展开

4